Lab 10
Device Trees

EELE 467
Due date: 11/19/2024

Embedded systems often have many external peripherals (LEDs, switches, buttons, Ethernet controllers, ADCs/DACs, etc.) attached via various buses (I2C, SPI, etc.). When an embedded Linux system boots, how does it know what hardware exists? We could hardcode the hardware details into the operating system and/or device drivers, but then we would have to do that for *every* embedded board we ever create—that would be a lot of work 😔. Instead, modern embedded Linux systems use a device tree, which is a data structure for describing hardware. From devicetree.org:

> "The devicetree is a data structure for describing hardware. Rather than hard coding every detail of a device into an operating system, many aspects of the hardware can be described in a data structure that is passed to the operating system at boot time."
>
> "A devicetree is a tree data structure with nodes that describe the devices in a system. Each node has property/value pairs that describe the characteristics of the device being represented. Each node has exactly one parent except for the root node, which has no parent."

We'll use device trees to inform Linux about our custom hardware (what memory address it's located at, how many registers it has, etc.). This enables our device drivers to interact with our custom hardware.

This lab relies heavily on the sysfs psuedo-filesystem. sysfs is many things, but it essentially provides access to kernel-related system information, e.g., device parameters, filesystems, etc. We will use files in sysfs to control the LEDs. For a brief overview, consult the sysfs(5) man page. We will gradually become more familiar with sysfs, but for now, we'll learn mostly by exploring the filesystem. One of the best ways to learn is to navigate the sysfs filesystem and see what's there; we encourage you to poke around the `/sys` directory on your virtual machine and/or the DE10-Nano.

> Before getting started, read through Linux and the Device Tree, Device Tree Usage, and textbook sections 9.4 through 9.4.2. Don't worry about understanding all the details just yet; skim the documents to get a general idea of how device trees work and what they look like.

Setting up our base device tree

Many devices trees are organized hierarchically by including other *device tree source include* (dtsi) files. For example, the Terasic DE0 Nano (the DE10 Nano's predecessor) device tree follows this pattern; it includes `socfpga_cyclone5.dtsi`, which then includes the base `socfpga.dtsi` file. As we move up the include-chain, the dts files often contain less information, and usually modify properties that were defined in the included file(s). Organizing device trees in this manner is a very useful abstraction.

We're going to create the following hierarchical device tree system:
`socfpga_cyclone5_led_patterns.dts` $\rightarrow$ `socfpga_cyclone5_de10_nano.dtsi` $\rightarrow$ `socfpga_cyclone5.dtsi` $\rightarrow$ `socfpga.dtsi`.

The DE10-Nano's predecessor, the DE0-Nano, has a device tree in the kernel source tree. Fortunately, these boards are basically the same, so we can use the DE0-Nano device tree as our starting point.

Creating the DE10 Nano include file

The DE0 device tree is located in the kernel source tree at

`linux-socfpga/arch/arm/boot/dts/intel/socfpga/socfpga_cyclone5_de0_nano_soc.dts.`

Step 1 Create the `socfpga_cyclone5_de10nano.dtsi` by copying the `socfpga_cyclone5_de0_nano_soc.dts` file:

```
cd <path-to-your-linux-socfpga-directory>/arch/arm/boot/dts/intel/socfpga
cp socfpga_cyclone5_{de0_nano_soc.dts,de10nano.dtsi}
```

The `cp` command above uses brace expansions to reduce the amount of typing we have to do. The brace expansion creates the following equivalent command: `cp socfpga_cyclone5_de0_nano_soc.dts socfpga_cyclone5_de10nano.dtsi`

Step 2 In the de10 nano dtsi file, change the `model` and `compatible` properties to reflect that the board is a DE10-Nano, not a DE0-Nano.

Creating the LED patterns device tree

Now we'll create out project-specific `socfpga_cyclone5_de10nano_led_patterns.dts` device tree. We want to keep the device tree source file in our git repository, but the file needs to be in the linux-socfpga repository as well so we can compile it. We'll use a symbolic link to make the file "appear" in both places.

Step 1 Create `socfpga_cyclone5_de10nano_led_patterns.dts` in your repository at `linux/dts/` with the contents shown in Listing 1:

```
#include "socfpga_cyclone5_de10nano.dtsi"

/{

};
```

Listing 1: Initial blank device tree source for our LED patterns project. We don't need anything in the device tree yet because we haven't written any device drivers for our LED patterns hardware.

Step 2 Create a symoblic link from `linux-socfpga/arch/arm/boot/dts/intel/socfpga/socfpga_cyclone5_de10nano_led_patterns` to the file in your repository:

```
ln -s <your-repo>/linux/dts/socfgpa_cyclone5_de10nano_led_patterns.dts
↪  <linux-socfpga>/arch/arm/boot/dts/intel/socfpga/
```

Use *full absolute file paths* for each of the arguments to `ln`.

This makes the dts file in `linux-socfpga/arch/arm/boot/dts/intel/socfpga/` point to the file in your repo. The dts file has to be in `linux-socfpga/arch/arm/boot/dts/intel/socfpga/` so the Linux kernel can compile it.

Step 3 There's a Makefile at `linux-socfpga/arch/arm/boot/dts/intel/socfpga/Makefile` that configures what device trees to compile. Add `socfpga_cyclone5_de10nano_led_patterns.dtb` to that Makefile.

Compiling the device tree

We're now ready to compile our device tree *source* into a device tree *blob*!

1. Navigate to your `linux-socfpga` directory
2. Build the dtb

   ```
   make ARCH=arm dtbs
   ```

3. Copy the `socfpga_cyclone5_de10nano_led_patterns.dtb` from `arch/arm/boot/dts/intel/socfpga` to `/srv/tftp/de10nano/led_patterns`; name the file in your tftp directory `led-patterns.dtb` or modify your led-patterns u-boot script to use `socfpga_cyclone5_de10nano_led_patterns.dtb` as the device tree blob name.

Verification
Reboot your DE10-Nano and make sure it boots using your new device tree.

Making sure you are using the new device tree

Let's make sure you are using the new device tree.

Verification
In PuTTY, navigate to `/proc/device-tree` and print out the `model` and `compatible` files. These should match the model and compatible properties in your device tree.

Note that the files don't end with a newline, so printing them with `cat` doesn't work very well. You can use `awk` instead: `awk 1 model`
`awk 1` somehow magically adds a newline to the output.

Controlling the HPS user LED on the DE10-Nano

The DE10-Nano has a user LED hooked up to the HPS. This LED is already in the device tree, so we can control the LED via the `gpio-leds` driver. In particular, we'll use the device attribute files that the `gpio-leds` driver exports to sysfs.

In `socfpga_cyclone5_de10nano.dtsi`, we are interested in the `leds` node, as shown in Listing 2.

We'll delve deeper into the meaning of the properties in Listing 2 later; for now, just remember that the LED is called `"hps_led0"`, as this is what it will be called in sysfs.

```
leds {
    compatible = "gpio-leds";
    led-hps0 {
        label = "hps_led0";
        gpios = <&portb 24 0>;
        linux,default-trigger = "heartbeat";
    };
};
```

Listing 2: leds device tree node. The **compatible** property specifies which driver(s) the device(s)/node(s) are compatible with. The `led-hps0` node describes the HPS user LED on the DE10-Nano; it is called `"hps_led0"`, is at pin offset 24 on `portb`, and is set to be a "heartbeat" (this option isn't enabled in the kernel by default, so it won't work out of the box for us). `portb` is defined in `socfpga.dtsi`.

Configuration via sysfs

Let's control the HPS user LED from the command line (in PuTTY). The LED is located near the Ethernet jack and pushbuttons on the DE10-Nano.

The LED's data structure will be located in `/sys/class/leds/hps_led0`. This directory is created by the gpio-leds driver; this driver was automatically *bound* to the `led-hps0` device via the **compatible** property.

Step 1 Navigate to the `/sys/class/leds/hps_led0` directory and type `ls -l` (or `ll` for short); you should see several files and directories, including `brightness`, `max_brightness`, and `trigger`. See what is stored in the files by `cat`'ing them.

`max_brightness` will be 255, but since the LED doesn't have brightness control (i.e. it doesn't support PWM), any brightness value ≥ 1 will turn the LED on.

You will need to be *root* to write to these sysfs files. The easiest thing to do is start a root shell by running `sudo -i`. After you do this, you'll be the root user and you'll be able to write to the sysfs attribute files.

Step 2 Play around by `echo`'ing values into the `brightness` file. You should be able to turn the LED on and off, e.g.

```
echo 255 > brightness
```

```
echo 0 > brightness
```

Step 3 Explore the `triger` file. `cat`'ing the file will show a bunch of different trigger options, and the active option will be shown in brackets. Unlike the `brightness` file, which accepts integers, the `trigger` file accepts a limited number of strings. For example, to activate the cpu trigger, you would type

```
echo cpu > trigger
```

Step 4 Use the `timer` trigger to create a heartbeat LED:

```
echo timer > trigger
```

Once you activate the `timer` trigger, two new files will show up: `delay_on` and `delay_off`. `cat` the `delay_on` and `delay_off` files. The values are in milliseconds, and should default to `500`. Change the `delay_on` and `delay_off` times to 1 second.

Demo
Show that you can turn the LED on/off and set different trigger sources.

Creating a heartbeat LED

As with any embedded system, we're going to add a "heartbeat" LED—because why not?

By now, you may have noticed that the default setting of `linux,default-trigger = "hearbeat"` didn't create a flashing LED. This is because the "heartbeat" LED trigger isn't enabled in the kernel by default.

Given that the "heartbeat" trigger doesn't work out of the box, there are two ways we can create a heartbeat LED:

1. Use a "timer" trigger to create a blinking LED. The timer trigger has more configuration possibilities than the heartbeat trigger.
2. Enable the "heartbeat" trigger in the kernel. The heartbeat trigger pulses twice, kind of like a heartbeat. The pulse rate increases with increasing system load, i.e., it pulses faster if the system is doing more processing—neat!

We're going to use the second option. We'll start by configuring and recompiling the kernel.

Kernel configuration

Remember to set the `ARCH=arm` environment variable when doing anything with the kernel, otherwise you'll start configuring things for x86, which will cause some headaches and make you have to reconfigure and recompile the kernel for ARM.

See textbook section 9.2.3 for some information on how to configure the kernel using menuconfig

Removing version information from the kernel version string

Making changes to the kernel repository changes the kernel's version string: the kernel appends the git repo's HEAD commit hash to the end, and it appends "dirty" when there are uncommitted changes. We have already made some changes in the `linux-socfpga` repository, so our compiled zImage will have version information appended to it.

Since your device drivers will be compiled against a specific kernel version, they contain the kernel's version string. When the version string doesn't match between your zImage and your drivers, you can't (easily) load your drivers—the kernel will complain about a "version magic" mismatch.

We can disable this behavior by disabling the "Automatically append version information to the version string" kernel configuration option. The option is located at *General setup* → *Automatically append version information to the version string*. Disabling this option will make the version string something like "5.10.100+" instead of "5.10.100-gfd86faf7bfd3".

Change that configuration option using `menuconfig` or `nconfig`:

```
make ARCH=arm menuconfig
```

```
make ARCH=arm nconfig
```

Enabling the heartbeat trigger

As mentioned before, the heartbeat trigger isn't enabled in the kernel by default, so we need to enable it and rebuild the kernel. The LED trigger options are located under `Device Drivers` → `LED Support` → `LED Triger support`. Turn on the heartbeat trigger using `menuconfig` or `nconfig`:

```
make ARCH=arm menuconfig
```

```
make ARCH=arm nconfig
```

Recompile the kernel

Cross-compile the kernel as you did previously in HW 8. See section 9.2.4 of the texbook for instructions on cross-compiling the kernel.

Once you've recompiled the kernel, copy the zImage to your tftp server:

```
cp <linux-socfpga>/arch/arm/boot/zImage /srv/tftp/de10nano/kernel
```

Verification

Reboot your FPGA and make sure the heartbeat LED works on boot.

Updating the device tree

Documentation for all device tree bindings, which specify node types and properties, are located in `Documentation/devicetree/bindings` in the kernel repo. You can browse the files using your local kernel repository, Intel's linux-socfpga repository on GitHub, or bootlin's Elixir Cross Referencer (Elixir is really fantastic for exploring the Linux kernel's codebase). If you use GitHub or Elixir to view the device tree bindings documentation, make sure you are browsing the kernel version that matches your kernel.

Device tree binding documentation used to be in free-form plain txt files. This format is now a legacy format. The newer standard uses YAML files. Not all device tree bindings documentation have been converted to YAML yet. YAML device tree schemas seem to still be pretty early on in development. If you're interested in reading more, see here and here. Device tree stuff in general is very much under active development—exciting times to be an embedded Linux person!

Have the leds-gpio, leds-common, and gpio device tree binding documentation files open, as those text files contain all the information we need to know about the properties we will be using.

Now that we've enabled the heartbeat trigger in the kernel, the heartbeat LED should be working. However, as noted in the leds-common documentation, the `label` property is deprecated; the documentation suggests using the `color` and `function` properties instead.

Step 1 Update the `led-hps0` LED node in `socfpga_cyclone5_de10nano.dtsi` as follows:

- Delete the `label` property from the `led-hps0` node.
- Add the `color` and `function` properties to the node. The `leds/common.h` header file has `#define` statements for common LED colors and functions, which we should use. Set the LED to be a green heartbeat LED. You'll need to add the following `#include` directive to your `socfpga_cyclone5_de10nano.dtsi` file: `#include <dt-bindings/leds/common.h>`

Step 2 Recompile your device tree and copy the dtb file to your tftp server. You may have to remove your compiled led patterns dtb file from `arch/arm/boot/dts/intel/socfpga` if recompiling doesn't do anything.

The LED's sysfs directory will now be named `green:heartbeat` instead of `hps_led0`

Demo
Demonstrate that the heartbeat LED works and that the LED's sysfs directory is named `green:heartbeat`

Deliverables

Demonstrations

1. Show that you can turn the LED on/off and set different trigger sources.
2. Show that the heartbeat LED works on boot.
3. Show that the LED's sysfs directory is named `green:heartbeat`.

GitHub Submission

Follow the workflow given in your repository (`docs/workflow.md`), per usual. Make sure all your files are committed.

Questions

Answer the following questions in your markdown file:

1. What is the purpose of a device tree?